CPA-DF: A Tool for Configurable Interval Analysis to Boost Program Verification

Dirk Beyer
LMU Munich, Germany

Po-Chun Chien
LMU Munich, Germany

Nian-Ze Lee
LMU Munich, Germany

***Abstract*—Software verification is challenging, and auxiliary program invariants are used to improve the effectiveness of verification approaches. For instance, the *k*-induction implementation in CPAchecker, an award-winning framework for program analysis, uses invariants produced by a configurable data-flow analysis to strengthen induction hypotheses. This invariant generator, CPA-DF, uses arithmetic expressions over intervals as its abstract domain and is able to prove some safe verification tasks alone. After extensively evaluating CPA-DF on SV-Benchmarks, the largest publicly available suite of C safety-verification tasks, we discover that its potential as a stand-alone analysis or a sub-analysis in a parallel portfolio for combined verification approaches has been significantly underestimated: (1) As a stand-alone analysis, CPA-DF finds almost as many proofs as the plain *k*-induction implementation without auxiliary invariants. (2) As a sub-analysis running in parallel to the plain *k*-induction implementation, CPA-DF boosts the portfolio verifier to solve a comparable amount of tasks as the heavily-optimized *k*-induction implementation with invariant injection. Our detailed analysis reveals that dynamic precision adjustment is crucial to the efficiency and effectiveness of CPA-DF. To generalize our results beyond CPAchecker, we use CoVeriTeam, a platform for cooperative verification, to compose three portfolio verifiers that execute CPA-DF and three other software verifiers in parallel, respectively. Surprisingly, running CPA-DF merely in parallel to these state-of-the-art tools further boosts the number of correct results up to more than 20 %.**

Demonstration video: https://youtu.be/l7UG-vhTL_4

I. Introduction

As the technology evolves, software systems are becoming increasingly complicated. Assuring the safety of these systems has always been a crucial research field, and numerous verification tools and algorithms have been proposed. To keep up with the development pace of software systems, *cooperative verification* [1] aims at combining the strengths of different tools to form a more powerful verifier. One prominent example is to augment *k*-induction with an invariant generator. Auxiliary invariants prune out unreachable program states and can strengthen the induction hypotheses, greatly improving the proof-finding capability of *k*-induction [2, 3, 4].

CPAchecker, an award-winning software analyzer, is one of the many tools implementing the cooperative *k*-induction method [3]. It uses a configurable data-flow analysis CPA-DF based on expressions over intervals to generate invariants. While profiling the performance of *k*-induction, we observed that CPA-DF is able to construct a proof, an inductive invariant that implies the safety property, on its own for many verification tasks. A question naturally arose: *How much of the performance boost by combining k-induction with* CPA-DF *actually comes from the parallel execution, but not from the invariant injection?* Surprisingly, we found out that by running the two analyses purely in parallel, we can solve almost as many tasks as running the two cooperatively (with invariant injection). In fact, nearly 90 % of the boosting effect of this cooperative approach in CPAchecker is achieved by the parallel execution. That is, without the need of a more complicated mechanism to communicate invariants, a simple portfolio is already very effective. This motivates us to further investigate (1) the capability of CPA-DF and (2) whether this boosting effect can be generalized to other tools. The main engineering challenge was to decouple the integrated invariant-generation component CPA-DF from *k*-induction and use it modularly with other components or tools. Thanks to the high flexibility and adaptivity of the CPA framework [5] and CoVeriTeam [6], we were able to overcome the challenge elegantly.

Use Cases. In this paper, we demonstrate two major use cases of CPA-DF: (1) as a stand-alone analyzer and (2) as a complementary performance booster in a portfolio-based verifier. We envision that CPA-DF could be beneficial to both verification-tool users and developers.

Novelty and Contributions. We discover a novel aspect of the existing invariant-generation component of CPAchecker and decouple it into a stand-alone static analyzer CPA-DF. The presented analyzer is highly configurable and can find a comparable amount of proofs as plain *k*-induction. To demonstrate the usability of the tool, we conducted a large-scale evaluation on more than 9 000 verification tasks. We show that in CPAchecker, running CPA-DF as a parallel component along with *k*-induction achieves a performance boost similar to that of a complicated cooperative approach. This boosting effect can also be observed by pairing CPA-DF with another well-established tool as a portfolio-based verifier. The finding implies that a parallel portfolio is a simple yet very effective way to improve performance.

Tool Availability. We contribute the open-source data-flow analysis tool CPA-DF (see Section "Data-Availability Statement").

II. Related Work

Our work is mainly related to data-flow analysis and cooperative verification.

Data-Flow Analysis. Data-flow analysis computes the information flow throughout the control-flow graph of a program. It is used by compilers for tracking the reaching definitions

DOI 10.1109/ASE56229.2023.00213

and performing constant propagation [7]. Depending on the used abstract domain, it can analyze different aspects of a program. Domains of explicit values [7] and intervals [8, 9] are commonly used to overapproximate numerical values of program variables. Compared to model checking, data-flow analysis is usually less precise but more efficient [10].

Cooperative Verification. Cooperative verification approaches [1] combine analyses with different strengths. Using auxiliary invariants to confine the state space of the main analysis has been studied for many algorithms, including *k*-induction [2, 3, 4], predicate abstraction [11, 12], and IC3/PDR [13]. In contrast to cooperative approaches demanding tight integration between analyses, verifiers can also be combined easily as a sequential or parallel portfolio.

III. Configurable Interval Analysis

Configurable Program Analysis with Precision. A *configurable program analysis with dynamic precision adjustment* (CPA+) [14] specifies an abstract domain and a set of precisions used to inspect a program. The precision is adjustable to make the analysis either efficient but coarse or precise but time-consuming. A CPA+ $\mathbb{D}$ together with an initial abstract state e_0 and a precision π is given to the CPA+ algorithm (Algorithm 2.1 in the referred publication [14]) to construct a set of reachable abstract states under precision π.

A CPA+ with Arithmetic Expressions over Intervals. CPA-DF relies on a CPA+ using arithmetic expressions over intervals as its abstract domain. This interval CPA+ $\mathbb{I}$ was first used as a generator of auxiliary invariants to boost the performance of *k*-induction [3]. The complete description of $\mathbb{I}$ can be found in the technical report [15]. We briefly summarize the major features of $\mathbb{I}$: In the abstract domain of $\mathbb{I}$, every program variable is mapped to an arithmetic expression over intervals, e.g., $[l_1, u_1] \cup [l_2, u_2]$, where l_i and u_i are numerical values representing the lower and upper bounds of an interval, respectively. Compared to a plain interval analysis that only tracks a single interval for each variable, $\mathbb{I}$ is able to represent complex ranges of variables and hence more precise. A precision π for $\mathbb{I}$ includes a set of *important variables* and a Boolean flag toggling whether *widening* [9] is applied or not. Two abstract states are merged only if their interval expressions match over all important variables, and the merged state will take the union of the interval expressions of the two abstract states. The widening operation further relaxes the abstraction of a merged state by computing the upper and lower bounds and assigning a single interval to each variable.

Dynamic Precision Adjustment. CPA-DF combines the interval CPA+ $\mathbb{I}$ with a precision-refinement strategy to achieve an efficient and effective data-flow analysis. The procedure is described in Alg. 1. It takes as input the interval CPA+ $\mathbb{I}$[1], an initial abstract state e_0 mapping every variable to $(-\infty, \infty)$ with an initial precision π_0 using the empty set of important variables and widening, and a safety property P. The CPA+

Algorithm 1 Interval analysis with precision refinement

```
Input: the interval CPA+ $\mathbb{I}$, an initial abstract state $e_0$, an
    initial precision $\pi_0$, and a safety property $P$
Output: safe if $P$ holds for all reachable abstract states, or
    unknown if the analysis is inconclusive
1: $\pi := \pi_0$
2: while $\pi \neq nil$ do
3:     $R :=$ CPA+$(\mathbb{I}, e_0, \pi)$
4:     if $\forall s \in R : s \models P$ then
5:         return safe
6:     $\pi :=$ RefinePrecision$(\mathbb{I}, \pi)$
7: return unknown
```

algorithm [14] is invoked to compute a set R of abstract reachable states under the given precision. If every abstract state in R satisfies P, the program is safe. Otherwise, it means that the analysis is not precise enough to determine the safety of the program. In this situation, $\mathbb{I}$ dynamically adjusts the precision by marking more variables as important via a heuristic and disabling widening (at line 6). The algorithm continues until either the program is proven to be safe under a refined precision, or precision adjustment is no longer possible, namely, all variables are marked as important, and widening is disabled. In this case, subroutine RefinePrecision() returns the special precision nil, and the analysis terminates as inconclusive.

IV. Experimental Evaluation

We demonstrate the two major use cases of CPA-DF presented in Sect. I by investigating the research questions below:

- **RQ1:** Can dynamic precision adjustment improve CPA-DF as a stand-alone analysis for finding proofs?
- **RQ2:** Can a parallel portfolio of CPA-DF and the plain *k*-induction implementation compete with the optimized implementation with invariant injection in CPAchecker?
- **RQ3:** Can CPA-DF complement other state-of-the-art software verifiers as a sub-analysis in a parallel portfolio?

Benchmark Set. We used the verification tasks from the 2023 Intl. Competition on Software Verification (SV-COMP '23). We selected only verification tasks whose safety property is the reachability of calls to an error function. The overall benchmark set consists of a total of 9537 verification tasks, where 3153 tasks contain a known specification violation, and the remaining 6386 tasks are considered safe.

Experimental Setup. All experiments were conducted on machines with a 3.4 GHz CPU (Intel Xeon E3-1230 v5, 8 processing units) and 33 GB of RAM. The operating system was 64-bit Ubuntu 22.04, running Linux 5.15 and OpenJDK 17.0.5. Each verification task was limited to 4 processing units, 15 min of CPU time, and 15 GB of RAM. We used BenchExec [16] for reliable benchmarking, revision 43678 on branch `data-flow-exp-configs` of CPAchecker for the implementations of CPA-DF and *k*-induction, and CoVeriTeam [6] at commit 838e67ff to compose parallel portfolios. As representative state-of-the-art tools, we chose Esbmc [17], Symbiotic [18], and UAutomizer [19], whose

[1]In the implementation, a composite CPA of $\mathbb{I}$ and other supportive CPAs tracking program locations, pointers, and call stacks is used.

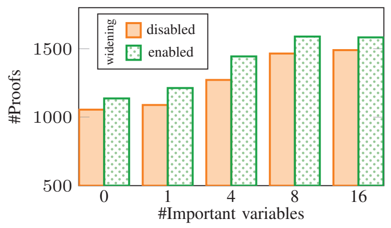

Fig. 1: Numbers of proofs found by CPA-DF with different precisions

TABLE I: Summary of the verification results on 9537 benchmark tasks

Framework		CPAchecker				CoVeriTeam					
Configuration	(tasks)	DF	KI	KI ‖ DF	KI↫DF	Esbmc	E. ‖ DF	Symbiotic	S. ‖ DF	UAutomizer	U. ‖ DF
Correct results	9537	1719	3258	4063	4164	5063	5201	3578	4419	3993	3983
proofs	6386	1719	1738	2572	2673	2962	3143	2046	2904	2930	2984
alarms	3153	0	1520	1491	1491	2101	2058	1532	1515	1063	999
Wrong proofs		0	0	0	15	17	17	0	0	0	0
Wrong alarms		0	2	2	2	10	10	1	1	0	0
Unknowns		6537	6277	5472	5356	4075	4309	5604	5117	5356	5554
Solved by DF		1719	-	877	956	-	359	-	859	-	1282

archives can be automatically downloaded and installed by CoVeriTeam from the SV-COMP '23 artifacts. The time limit of CPA-DF was set to 5 min in parallel portfolios to allocate enough time to the main analysis.

RQ1: Dynamic Precision Adjustment in CPA-DF. To study the effect of dynamic precision adjustment on the proof-finding capability of CPA-DF, we executed it with fixed precisions limiting the number of important variables to $\{0, 1, 4, 8, 16\}$ and switched the widening operation on and off. Figure 1 shows the number of safe tasks that CPA-DF proved under each static precision, and the results of CPA-DF with dynamic precision adjustment are shown in column "DF" of Table I.

In Fig. 1, we observe that more proofs were found as the number of important variables increases. However, marking too many variables as important is suboptimal because this will prevent abstract states from being merged: The analysis will need to process more abstract states and thus might not finish timely (details below). For the widening operation, we also observe a similar phenomenon: Without the overapproximation performed by widening, the analysis found fewer proofs.

The scatter plot in Fig. 2 compares the CPU time CPA-DF used to find a proof under the precisions enabling widening and allowing 0 and 16 important variables, respectively. The two precisions are denoted as $(0, t)$ and $(16, t)$. A data point (x, y) in the plot means that there is a task solvable by CPA-DF under both precisions, where precision $(0, t)$ requires y seconds, and precision $(16, t)$ requires x seconds. In this comparison, although CPA-DF with precision $(0, t)$ found fewer proofs than precision $(16, t)$ (1137 vs. 1583), it often found proofs faster for the tasks solvable with both precisions. To automatically search for a suitable precision for a verification task, CPA-DF dynamically refines the precision from the lowest one $(0, t)$ to the highest (N, f), where N is the number of all variables in the program under analysis. In this way, CPA-DF is as precise as necessary and as efficient as possible. Overall, CPA-DF with dynamic precision adjustment solved more tasks than any other static precision in our experiments (column "DF" of Table I). Notably, CPA-DF proved the correctness of many tasks arising from practical programs, including device drivers for Linux (Category *SoftwareSystems-LDV*) and software product lines (Category *ReachSafety-ProductLines*).

RQ2: Parallel Portfolio versus Invariant Injection. To compare the effect on k-induction between running CPA-DF as a sub-analysis in a parallel portfolio and as a generator for invariant injection, we evaluated four different analyses in CPAchecker: the configurable interval analysis described in this paper (DF), the plain k-induction (KI) without auxiliary invariant, the parallel portfolio of KI and DF (KI ‖ DF), and KI with induction hypotheses strengthened by auxiliary invariants injected from DF (KI↫DF) [3].

The experimental results are summarized in Table I. Additionally, Fig. 3 shows a quantile plot comparing the proof-finding capabilities of the assessed methods. A data point (x, y) in the plot indicates that x safe tasks are correctly proved by the respective algorithm within a CPU time bound of y seconds each. Observe that by combining DF with KI, either with communication (KI↫DF) or without (KI ‖ DF), we can achieve a significant improvement over running a single analysis, meaning that the two analyses complement each other.

We also note that although KI ‖ DF did not prove as many tasks as KI↫DF, it avoided the 15 wrong proofs[2] caused by invariant injection. All of these wrong proofs came from tasks in the *ReachSafety-Hardware* category, where KI↫DF also found most additional proofs over KI ‖ DF. In other words, the extra proofs found by KI↫DF in this category might be due to unsound overapproximation. If we exclude the tasks in this category, KI ‖ DF and KI↫DF found 2551 and 2554 proofs in the remaining tasks, respectively. Our results show that a parallel portfolio of CPA-DF and k-induction already brings us most performance improvement that can be gained with invariant injection, which involves a complicated information exchange between analyses and is more error-prone.

RQ3: Boosting Program Verification with CPA-DF. To examine the boosting effect of CPA-DF on other tools, we paired Esbmc, Symbiotic, and UAutomizer from SV-COMP '23 with CPA-DF via CoVeriTeam to form three parallel portfolios, respectively. Table I contains the results of running the verifiers alone and in parallel with CPA-DF. A quantile plot comparing the performance of these tool combinations on the safe verification tasks is also shown in Fig. 4. At the expense of discovering fewer violations because of the CPU time spent by CPA-DF, all three verifiers were able to find more proofs via having CPA-DF as a parallel component[3]. The extent of improvement depends on how much the sets of solvable tasks by the sub-analyses in the portfolio overlap. For example, there

[2]The issue (https://gitlab.com/sosy-lab/software/cpachecker/-/issues/1070) has been discoverd by the community and is under investigation.

[3]Even though UAutomizer was able to solve more tasks than UAutomizer ‖ DF overall, if we apply the scoring scheme from SV-COMP, which assigns two (resp. one) points to each correct proof (resp. alarm), UAutomizer ‖ DF obtains a higher score than UAutomizer.

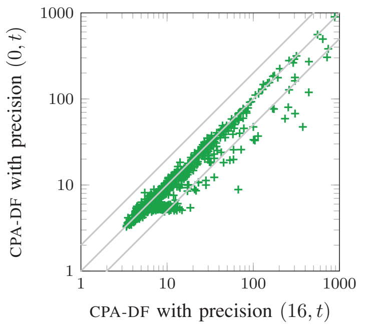

Fig. 2: CPU-time scatter plot of precisions $(16, t)$ and $(0, t)$

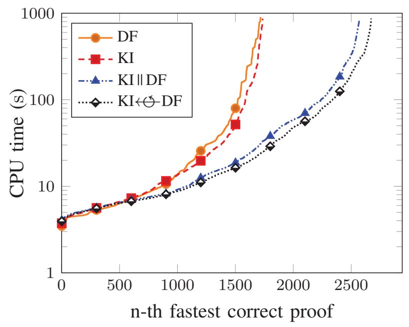

Fig. 3: Quantile plot comparing DF, KI, and their combinations in CPAchecker

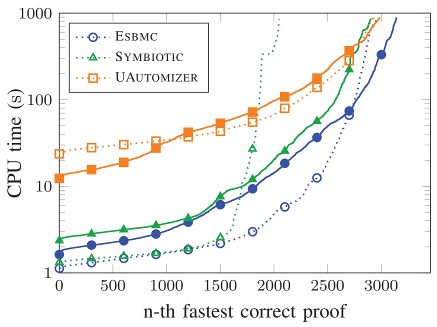

Fig. 4: Quantile plot showing the boosting effect on others (hollow: w/o DF, solid: w/ DF)

were 1530 tasks solvable by both UAutomizer and CPA-DF, which already accounts for 89 % of the tasks the latter could solve. Such a large overlap explains why UAutomizer ∥ DF only had a marginal improvement over UAutomizer. However, 1282 (43 %) proofs were found by CPA-DF in the portfolio, meaning that CPA-DF was able to solve these tasks faster than UAutomizer. By contrast, Symbiotic benefited the most from running CPA-DF in parallel because the solvable sets of the two sub-analyses are more disjoint (only 767 tasks were solvable by both). From Fig. 4, we can also observe a more significant improvement on Symbiotic than the other two verifiers.

V. Conclusion

We presented CPA-DF, a data-flow analysis tool based on interval expressions, originally developed as an auxiliary invariant generator integrated with the *k*-induction implementation in CPAchecker. Decoupling CPA-DF as a stand-alone tool, we are able to execute it modularly with other tools. In our evaluation, we observed that executing CPA-DF in parallel to a main analysis is not only helpful for *k*-induction in CPAchecker, which achieved a comparable performance as invariant injection, but also enhances other state-of-the-art verifiers. We envision verifier developers and users trying out CPA-DF to boost the performance of their tools. For future work, we plan to export invariants derived by CPA-DF in an exchangeable format to broaden its application.

Data-Availability Statement. CPA-DF is implemented in CPAchecker, available on SVN (https://svn.sosy-lab.org/software/cpachecker/branches/data-flow-exp-configs@43678, sources) and on Zenodo [20] (executable, for reproduction). More information and interactive tables for convenient browsing of the results are available at https://www.sosy-lab.org/research/cpa-df/.

Funding Statement. This project was funded in part by the Deutsche Forschungsgem. (DFG) – 378803395 (ConVeY).

References

[1] Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. Springer (2020). doi:10.1007/978-3-030-61362-4_8

[3] Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. Springer (2015). doi:10.1007/978-3-319-21690-4_42

[2] Donaldson, A.F., Haller, L., Kröning, D.: Strengthening induction-based race checking with lightweight static analysis. In: Proc. VMCAI. pp. 169–183. Springer (2011). doi:10.1007/978-3-642-18275-4_13

[4] Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. pp. 145–161. Springer (2015). doi:10.1007/978-3-662-48288-9_9

[5] Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. Springer (2007). doi:10.1007/978-3-540-73368-3_51

[6] Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. Springer (2022). doi:10.1007/978-3-030-99524-9_31

[7] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)

[8] Harrison, W.H.: Compiler analysis of the value ranges for variables. IEEE Trans. Softw. Eng. **SE-3**(3), 243–250 (1977). doi:10.1109/TSE.1977.231133

[9] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL. pp. 238–252. ACM (1977). doi:10.1145/512950.512973

[10] Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). doi:10.1007/978-3-319-10575-8_16

[11] Fischer, J., Jhala, R., Majumdar, R.: Joining data flow with predicates. In: Proc. FSE. pp. 227–236. ACM (2005). doi:10.1145/1081706.1081742

[12] Jain, H., Ivancic, F., Gupta, A., Shlyakhter, I., Wang, C.: Using statically computed invariants inside the predicate abstraction and refinement loop. In: Proc. CAV. pp. 137–151. Springer (2006). doi:10.1007/11817963_15

[13] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV. pp. 343–361. Springer (2015). doi:10.1007/978-3-319-21690-4_20

[14] Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE. pp. 29–38. IEEE (2008). doi:10.1109/ASE.2008.13

[15] Beyer, D., Dangl, M., Wendler, P.: Combining k-induction with continuously-refined invariants. Tech. Rep. MIP-1503, University of Passau (January 2015). doi:10.48550/arXiv.1502.00096

[16] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). doi:10.1007/s10009-017-0469-y

[17] Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: Proc. ASE. pp. 888–891. ACM (2018). doi:10.1145/3238147.3240481

[18] Slabý, J., Strejček, J., Trtík, M.: Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In: Proc. FMICS. pp. 207–221. Springer (2012). doi:10.1007/978-3-642-32469-7_14

[19] Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. Springer (2013). doi:10.1007/978-3-642-39799-8_2

[20] Beyer, D., Chien, P.C., Lee, N.Z.: Reproduction package for ASE 2023 article 'CPA-DF: A tool for configurable interval analysis to boost program verification'. Zenodo (2023). doi:10.5281/zenodo.8245821